Verifying Classic McEliece: examining the role of formal methods in post-quantum cryptography standardisation

Martin Brain[3], Carlos Cid[2,4], Rachel Player[1], and Wrenna Robson[1]

[1] Royal Holloway University of London, Egham, UK
[2] Simula UiB, Bergen, Norway
[3] City, University of London, Northampton Square, London, UK
[4] Okinawa Institute of Science and Technology Graduate University, Okinawa, Japan

Abstract. Developers of computer-aided cryptographic tools are optimistic that formal methods will become a vital part of developing new cryptographic systems. We study the use of such tools to specify and verify the implementation of Classic McEliece, one of the code-based cryptography candidates in the fourth round of the NIST Post-Quantum standardisation Process. From our case study we draw conclusions about the practical applicability of these methods to the development of novel cryptography.

Acknowledgements Wrenna Robson was supported by the EPSRC and the UK Government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/S021817/1).

1 Introduction

Computer-aided cryptography [10] is the field of study which "develops and applies formal, machine-checkable approaches to the design, analysis, and implementation of cryptography". This can be categorised into three strands:

- Establishing security guarantees at the design level, using symbolic and computational approaches.
- Verifying that implementations (new or pre-existing) are both efficient and functionally correct, by showing they conform to the design about which security guarantees have been established.
- Establishing security guarantees at the implementation level, such as constant-time execution and secret-data-independent memory accesses, both of which indicate resistance to timing attacks.

Panelists in a recent roundtable [29] on computer-aided cryptography expressed broad optimism on the future of the field, citing the success of tools and projects like HACL* [32], Fiat Cryptography [24], and Cryptoline [26]. One participant expressed the view that, within a few years, "the state of the art

in program proofs will have advanced enough that verifying primitives will be considered mundane and a strong requirement for any new proposed algorithm".

Despite this optimism, most submissions to the NIST Post-Quantum Cryptography standardisation Process [5] made no documented use of computer-aided cryptography in their development. Indeed, only two submissions — NTRU Prime [15] and Classic McEliece [8] — made mention of any potential use of formal methods in improving their designs. The NTRU Prime supporting documentation stated [15] that the design choices of the scheme enabled easier formal verification of its security properties, and that the authors had begun work on verifying the optimised NTRU Prime implementation against the reference implementation [14]. The Classic McEliece specification suggested a need for formally verified proofs of quantum security, and also mentioned the potential of formal verification of defences against timing attacks. Moreover, there has not been much use of computer-aided formal techniques in the evaluation of any of the schemes proposed for standardisation thus far.

Recently, NIST concluded the third round of their standardisation process. After round three, none of the remaining code-based candidates were selected for standardisation, but all of them were moved forward to the fourth round [1]. The isogeny-based scheme SIKE was also advanced to the fourth round, but has subsequently seen a successful attack on its underlying hard problem [21]. If any fourth-round candidates for KEM are selected for standardisation, they are thus likely to be a code-based, which motivates further scrutiny of these candidates.

The security of the remaining code-based candidates is reasonably well-understood, especially Classic McEliece, which has been long studied. Therefore, other criteria will play an important role in evaluating and distinguishing these schemes. We argue that applying the tools of computer-aided cryptography to study these schemes is vital at this stage. Firstly, the amenability of each scheme to being scrutinised and verified using these tools could be a criterion for their evaluation. Secondly, demonstrating that the design or an implementation of a scheme has been verified gives further confidence in this scheme.

In this work, we focus on applying computer-aided cryptography techniques for developing efficient verified implementations to the Classic McEliece scheme. Our main focus is an application of the SAW/Cryptol toolchain [25, 20] to the Classic McEliece reference implementation. We also report on our recent efforts using the interactive theorem prover Lean in the verification of the mathematics underlying aspects of the Classic McEliece design.

1.1 Related work

Verification of code-based cryptography To the best of our knowledge, there are only two other works [4, 3] whose goal, as in this work, is to create a formal specification of Classic McEliece. The first is a partial specification [4] written in Cryptol, which, as far as we are aware, has not been used for the verification of an implementation. The specification is comparable in size to the one we produced in this work, but is incomplete in different ways. Moreover, it does not seem to correspond to a named version of the Classic McEliece

implementation, and targets a different parameter set that the one we aimed at. Thus, we did not derive our own specification from it.

The second is a specification [3] written in Lean 4, that was made public only after this work was concluded. As with our work, the goal of that work was to investigate the use of Lean 4 for cryptographic specification. The specification [3] compiled to an implementation of the Round 3 version of Classic McEliece which passes the Known Answer Tests, and some of its functions have proven properties.

Considering the formalisation of code-based cryptography more broadly, the HOL Light theorem prover was used in [13] to formally verify an algorithm to calculate the "control-bits" of a permutation. The implementation of this algorithm is a key component of Classic McEliece. We also note that there are formalisations in Coq of linear error-correcting codes [7], but they are not orientated towards cryptography.

Applications of the SAW/Cryptol toolchain While our application of the SAW/Cryptol toolchain to Classic McEliece is novel, we draw inspiration from prior work applying this toolchain to other cryptographic schemes. The primary work we build on is a work from Galois and Amazon [19]. The paper takes two highly-optimised, trusted implementations of two current primitives, AES-256-GCM and SHA-384, and describes the process of proof engineering and tool development that led to high-level functional correctness proofs for these primitives. It demonstrates some of the current capabilities and limits of the toolchain; as we will discuss, our own work demonstrates different limits.

Developing verified implementations While our work focuses on verifying an existing implementation, another approach [9, 18] is to generate verified code directly from the specification. For example, a framework for building verified cryptographic implementations is provided in [9]; delivering assembly code that is provably functionally correct, protected against side-channels, and as efficient as hand-written assembly. The framework is illustrated in [9] by an application to the ChaCha20-Poly1305 cipher suite.

1.2 Our contributions towards Classic McEliece implementation verification

In our work, our primary goal is explore to what extent the Classic McEliece reference implementation submitted to the NIST standardisation process can be formally specified and verified using the SAW/Cryptol toolchain. Our secondary goal is to see if it is possible to find improvements on the implementation that make it easier to specify and verify, while not impacting performance or its security properties.

To this end, we offer in this paper:

- Formal specifications and verification proofs for large parts of the Classic McEliece reference implementation as of the Round 3 submission, available at [31].
- A revised implementation for one of the core encryption routines in Classic McEliece which both runs faster and admits a verification proof.
- A set of recommendations for designers and those setting standardisation criteria for engaging with computer-aided cryptography.

As an additional contribution, we also report on recent efforts to apply the interactive theorem prover Lean [27] [30] to produce verified proofs of certain mathematical constructions used in the design of Classic McEliece.

Our attack model assumes that an attacker has full access to the implementation source code, and is acting as a man-in-the-middle between a client and server attempting to perform key agreement using this KEM. Bugs in the implementation are relevant only when they cause the implementation to deviate from the theoretical design sufficiently that the IND-CCA2 security guaranteed by the design is violated. Deviations between the specification and implementation that do not cause a weakening of security in practice are less relevant. Our verification target, therefore, was the equivalence of parts of the Classic McEliece implementation with equivalent parts of the design.

2 Our toolchain and its target

The SAW/Cryptol toolchain consists of Cryptol [25], a domain-specific language for specifying cryptographic algorithms, and the Software Access Workbench (SAW) [20], a tool for verifying compiled bytecode against specifications defined using Cryptol. They have both been developed, and are maintained, by Galois Inc. Cryptol is a size-polymorphic and strongly typed functional programming language, and a Cryptol specification of an algorithm can resemble its mathematical specification more closely than an implementation in a general purpose language. The SAW/Cryptol toolchain is suitable for verifying pre-existing code against a specification. Thus, it could be applied to code that is highly trusted and so cannot be changed. The SAW/Cryptol toolchain could also form part of a continuous integration framework, where changes to the underlying program are run against tests that include SAW proof scripts.

The Classic McEliece key-establishment scheme is derived from the code-based public-key cryptosystem introduced in 1978 by McEliece [28]. The public key in the McEliece cryptosystem specifies a random binary Goppa code — a linear binary code with certain useful mathematical properties. The private key contains information necessary to perform efficient decoding from an input that is within a bounded error of a codeword. It is the claim of the designers that Classic McEliece is not "new" in any sense: it aims to be a conservative implementation of an established scheme. Where there are relatively novel elements, such as the storage of permutations in the form of control bits, they are not core to the PKE encryption and decryption operations.

Our target system for verification was the reference implementation for the version of Classic McEliece submitted in Round 3 of the NIST process [8], using the `mceliece348864` parameter set (the smallest parameter set). This reference implementation was created by the Classic McEliece team to be the definitive reference for their scheme's operation. It formed a key part of their submission to the NIST process, and has been available for manual public review since its initial release. Its operation and the rationale for its design have been fairly well-documented in the literature by its creators [16] [22], as have the operation of the optimised implementations released alongside it. Our decision to target the reference implementation was partly based on its status as a "golden reference" for the scheme, and partly because as a reference implementation we hoped to avoid verifying too much low-level optimisation, which can be extremely challenging [19].

We aimed to create a specification for this system in Cryptol, using SAW to create proofs of equivalency between the abstract Cryptol specification and the compiled bytecode of key functions used in the implementation. It should be noted that SAW and Cryptol were not originally designed for specifying and verifying asymmetric cryptography, but rather block ciphers, hash functions, and other forms of symmetric cryptography. As such, Cryptol is not a very expressive language for describing complex algebraic constructions, and specifying a scheme in this way can lead to unusably slow performance. On the other hand, expressing asymmetric ciphers in terms of bitwise operations as might suit Cryptol better could obscure the rich mathematical structures that often underlie them. This issue is not specific to Cryptol and reflects a general pattern in the computer-aided cryptography literature: it is a lot easier to find verifications of symmetric cryptographic schemes and implementations. We therefore expected that the task of applying the SAW/Cryptol toolchain to Classic McEliece to be very challenging, and this is why our primary goal was to test the capability limits of SAW/Cryptol in this task.

We emphasise that our SAW/Cryptol work sits in the 'second strand' of research in computer-aided cryptography (see Section 1). It does not cover aspects in the 'first strand' (such as formal verification of the Classic McEliece security proofs) or the 'third strand' (such as verifying that the implementations possess the claimed resistance to constant-time attacks).

3 Verifying Classic McEliece with SAW/Cryptol

We begin with an overview of what was successfully verified and what was not. Discounting those functions that are part of the RNG or that call the external hash function used in Classic McEliece, there are 41 different functions that make up the `mceliece348864` reference implementation. Of these, 18 were completely specified and verified; a further five were partially verified. All verifications were performed within WSL2 on a Windows 10 PC with 16GB of RAM and an Intel i5-8400 2.8Ghz processor. The limit on performance was generally memory: it is conceivable that, in a few cases, with a higher-specification machine it would

have been possible for some additional proofs to complete, but this would require further testing to verify. This work represents around 5.5 months of person-time.

In general, the lower-level "utility" functions were easily verifiable. The most substantial successes on cryptographically-relevant functions were in verifying the implementation of the finite field operations, both against a mathematically-defined specification and a literal translation of the C source, which we could prove were equivalent under relevant preconditions. The PKE encryption and decryption functions were where we found partial success, with full verification of the key calculation loops. We found little success with the KEM wrapper, which needed functions that Cryptol could not implement efficiently, and so it was impractical to test them.

Of the five functions that we partially verified, these can be split into three categories. Firstly, there were two functions for which it was possible to verify the core loop used in the function but not the function's effect across all loops. Secondly, there were two functions for which we found errors in the original code. These were the two sorting functions, which contain small but vital bugs in their comparators. Our discovery of these issues — which do not impact Classic McEliece directly but are nevertheless real bugs in these functions — demonstrates that verification can find problems that current methods of scrutiny might miss. Finally, there was one function that we found we could rewrite into a form that appears both slightly higher in performance and possible to verify. This function is especially interesting as it demonstrates that it is possible to write implementations that are "more verifiable", and this does not have to come at the cost of performance (in this case, quite the opposite) or, indeed, security.

The above information is summarised in Figure 1, which shows the call graph of functions in the implementation, with functions and their groupings colour-coded to signify the level of verification achieved. Green denotes that the contents of that box have been fully specified and verified, orange that the contents have been partially verified, red that the contents have been verified and a bug found, purple that a refactored version of the contents has been verified, and blue that the contents have not been specified or verified. A standalone version of this diagram is available at `https://github.com/linesthatinterlace/verifying-cmce/blob/main/docs/graph.pdf`

It should also be noted that all functions for which it was possible to symbolically execute them using SAW — which include all verified functions but also some that could not be fully verified — are thus guaranteed to be memory-safe and free of undefined behaviour, simply by virtue of having been executed through SAW's internal model.

3.1 Verification details

In the following, we highlight certain representative examples to demonstrate the different classes of function we considered. These are chosen in order to illustrate the strengths and weaknesses of the tools we used. Details of the specifications and proof scripts are omitted here but are available in a public repository [6].

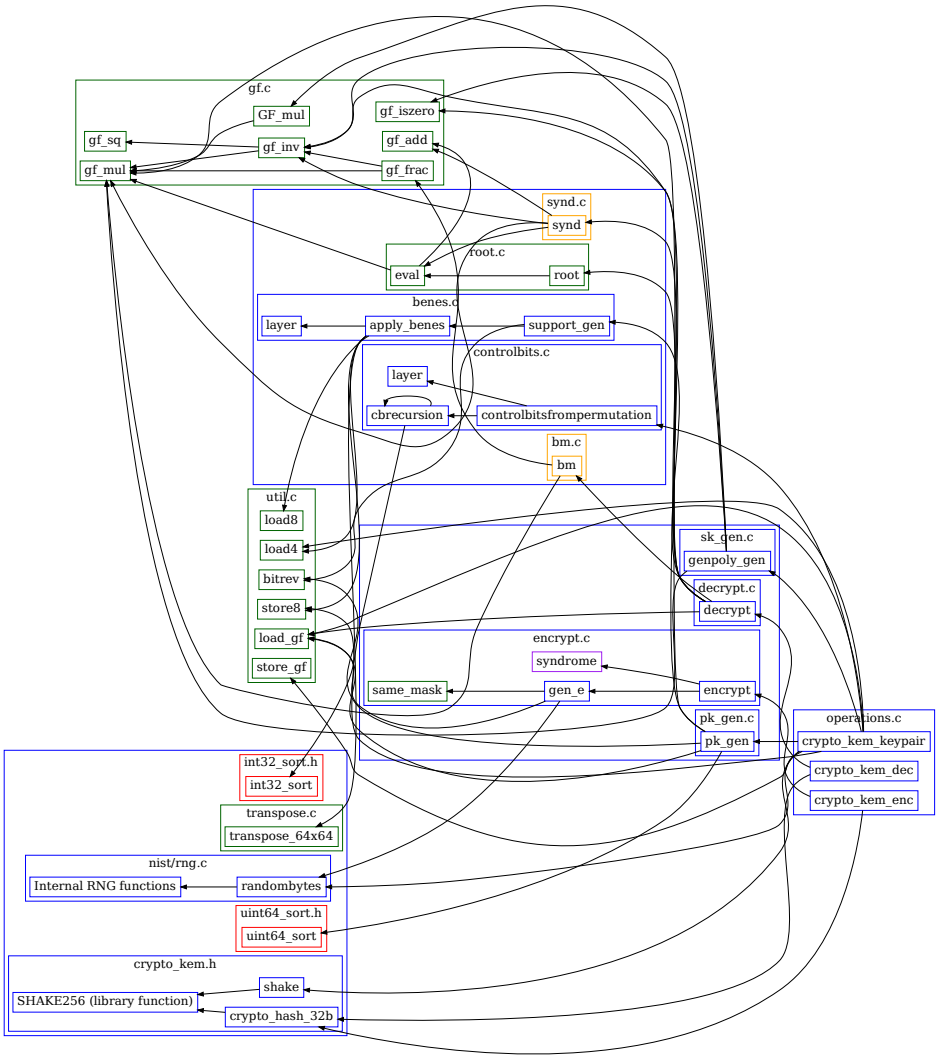

Fig. 1: The call graph of the Classic McEliece implementation with verification status highlighted.

Transposition The function `transpose_64x64` transposes a 64×64 matrix over the binary field $\mathbb{F}_2$, represented as a sequence of 64 64-bit unsigned integers taken as little-endian bitstrings. In the reference implementation, this is 33 lines-of-code, with the method of its operation being non-obvious, using masks to avoid any branching. We produced a two-line specification for this. This specification exactly captures the mathematical definition of `transpose_64x64`, adjusting for Cryptol's big-endian representation of 64-bit integers. There were multiple functions with simple definitions that had complex implementations (often to avoid branching for constant-time reasons), and these often admitted simple specifications against which they could be easily verified. The SAW/Cryptol toolchain excels at handling situations like these.

Field operations There is a set of low-level functions for performing operations over a finite field of prime order and one of its extensions. At the lowest level, these use simple bit operations, but for an operation like field division, a combination of field inversion and field multiplication is needed. SAW/Cryptol has the facility for compositional proofs, which means it can use equivalence proofs for functions that have already been proved or assumed in order to simplify a new proof that uses those functions. This lends itself well to proving facts about field arithmetic, where this is a common pattern.

For many of these field functions, we created two specifications: a low-level specification corresponding to the original implementation, and a higher-level one closer to the arithmetic implementation. The higher-level one is only equivalent to the low-level one under the condition that the inputs are valid members of the field. That this invariant is preserved — that is, that valid field members remain valid field members under all field operations — is a fact that it is possible and desirable to prove, but which was slightly out of scope of our own work (though we were able to do so in some cases). We were able to prove this equivalence under the assumption of this invariant using both SAW's symbolic execution and the native SMT support within Cryptol.

Loop functions There were some functions that took the form of loops operating on a large array. For these, it was possible to prove that the body of the loop (or a large section of it) was equivalent to the specification, but it was not possible to then use that equivalence to prove equivalence across all loop iterations.

We theorised that it might be possible to re-write one of these functions in some cases, re-factoring it to avoid one or more of the issues that cause SAW to stumble. In particular, a large amount of state needed to be carried across the loops — the entire current state of the array. We theorised that if the data flow could be isolated and split into smaller operations — removing the need to carry a large state — it would result in a more verifiable implementation. This turned out to be correct in at least one case.

The core component of encryption under Classic McEliece is the multiplication, in the ring of matrices over $\mathbb{F}_2$, of a vector e of length n by an $(n-k) \times n$

matrix H, producing the result $s := He$ of length $(n-k)$; this vector s is the *syndrome* of e. The values n and k are parameters of the scheme. For a given parameter set, n is the code length and k the code dimension of the underlying permuted Goppa code, the linear code on which Classic McEliece is based. H has the form $(I_{n-k} \mid T)$, where I_{n-k} is the identity matrix of dimension $(n-k)$, and T, the public key, is a $(n-k) \times k$ matrix.

For all parameter sets, n and k divide by 8, and thus so too does $n-k$. The data for e, T, and s are stored in the implementation as byte arrays, treating bytes as little-endian as with `transpose_64x64`. These byte arrays have length $\frac{n}{8}$, $(n-k) \times \frac{k}{8}$, and $\frac{(n-k)}{8}$ respectively.

The implementation version of the function `syndrome` takes the byte array storing T and e and the pointer of the byte array that will store s, reconstructs H, and loops over its $(n-k)$ rows. Each iteration of the loop sets a bit in s, performing the appropriate row multiplication of H for the current index with e. This is done using bit-manipulation to avoid branching.

The issue for verification is that this means each loop stores the entire state of s. We found that while we could verify that one loop correctly modified s per the specification, the symbolic execution of all of `syndrome` could not complete when we applied the compositional override. As each loop sets the appropriate bit of s by performing an accumulative OR on the appropriate byte with a suitably shifted weight-1 byte, every iteration was storing more and more accumulative information about the byte array s that SAW had to keep track of in the execution. We suspect that the loop became too large for SAW to support.

However, we were able to rewrite `syndrome` to avoid this issue. The key insight was the following one: each eight rows, a byte in s is completely determined, and thereafter plays no role in what follows. This is easy to see as a human reader, but is not information that SAW can derive. This is possibly because the index of the byte set each loop is calculated each time as $\frac{i}{8}$, where in our loop function we simply model the row index i as a 16-bit integer. It is possible that SAW, internally, cannot conclude that this implies that as i is incremented by one each time, after eight values of i a byte in s is fixed. Our idea was to implement not only a row-multiplication function, but also a function to take a block of eight rows and perform the corresponding multiplications across all of them, producing the resultant byte.

In addition, we noted that because e is multiplied by the identity matrix, it can be split into $(e_{id}, e_{pk}) := e$, where e_{id} and e_{pk} are stored as byte arrays of length $\frac{n-k}{8}$ and $\frac{k}{8}$ respectively. This means that $s = e_{id} + Te_{pk}$, and so we can focus on performing Te_{pk}, with no need to reconstruct H. We can calculate Te_{pk} a byte at a time, exclusive-or this byte to the corresponding byte of e_{id}, and this gives the corresponding byte of s.

This means that we are performing $\frac{n-k}{8}$ loops rather than $(n-k)$, but at each loop we perform eight row multiplications instead of one, albeit also on a smaller matrix. The crucial part is that then each loop sets a byte of s without reference to the current state of s. This is unlike the original implementation, which modifies s each loop by performing an OR operation on one of its bytes.

```
void syndrome(unsigned char *s, const unsigned char *pk,
  unsigned char *e)
{ unsigned char b, row[SYS_N/8];
  const unsigned char *pk_ptr = pk;

  int i, j;

  for (i = 0; i < SYND_BYTES; i++)
    s[i] = 0;

  for (i = 0; i < PK_NROWS; i++)
  { for (j = 0; j < SYS_N/8; j++)
      row[j] = 0;

    for (j = 0; j < PK_ROW_BYTES; j++)
      row[ SYS_N/8 - PK_ROW_BYTES + j ] = pk_ptr[j];

    row[i/8] |= 1 << (i%8);

    b = 0;
    for (j = 0; j < SYS_N/8; j++)
      b ^= row[j] & e[j];

    b ^= b >> 4;
    b ^= b >> 2;
    b ^= b >> 1;
    b &= 1;

    s[ i/8 ] |= (b << (i%8));

    pk_ptr += PK_ROW_BYTES; } }

```

Listing 1: Original `syndrome` implementation.

Not only does this implementation of `syndrome` actually allow efficient symbolic execution and a verification proof, but our re-implementation appears to result in a modest performance improvement over the original. In testing on random data, we found it ran in an average of 1.4ms as opposed to an average of 1.5ms for the original implementation. We should note that this implementation is also used in the scheme's optimised implementation, though not the additionally optimised implementation that uses assembly-level code. In addition, we believe we have maintained the side-channel resistance properties of the original – in testing, there did not appear to be an appreciable difference in performance with different random inputs, and there is no branching or memory accesses indexed with secret data.

```
unsigned char bytes_bit_dotprod(const unsigned char *u,
  const unsigned char *v, size_t n)
{ unsigned char b;
  int i;
  b = 0;
  for (i = 0; i < n; i++)
    b ^= u[i] & v[i];

  return b_func(b); }

unsigned char bytes_bit_mul_block(const unsigned char *u,
  const unsigned char *v, size_t n)
{ const unsigned char *u_ptr = u;
  unsigned char b;
  int i;
  b = 0;
  for (i = 0; i < 8; i++)
  {
    b += (bytes_bit_dotprod(u_ptr, v, n) << i);
    u_ptr += n;
  }

  return b; }

void syndrome_bytewise(unsigned char *s, const unsigned
  char *pk, unsigned char *e)
{ const unsigned char *pk_ptr = pk;
  const unsigned char *eid = e;
  const unsigned char *epk = e + SYND_BYTES;
  int i;
  for (i = 0; i < SYND_BYTES; i++)
  {
    s[i] = eid[i] ^ bytes_bit_mul_block(pk_ptr, epk,
  PK_ROW_BYTES);
    pk_ptr += 8*PK_ROW_BYTES;
  } }

```

Listing 2: Revised `syndrome` implementation.

In summary, we were able to produce an implementation of `syndrome` that was faster, as secure, and, crucially, more verifiable than the original. This is evidence for the following claim: it is meaningful to consider one implementation being more verifiable than another, and there are properties a function can have that make it inherently easier to find a more verifiable implementation. If it was not the case that the data flow for `syndrome` could be separated as we did here, then the above approach would not have worked. Thus, at the design stage, such "separability" could be considered as a desirable attribute that can make an implementation more verifiable (for some chosen definition of verifiability).

Sorting comparator bugs We discovered bugs in the sorting functions used by the Classic McEliece implementation, which do not directly affect the implementation but could certainly present an issue in code reuse. The bugs we found were in macros used by the sorting functions as element comparators. These macros take two variables, and place the minimum value in the first variable and the maximum one in the second without performing a branch. The first acts on two unsigned 64-bit integers, and the second acts on two signed 32-bit integers. In the former case, it simply does not produce the right output for certain inputs. In the latter, signed integer overflow can occur for certain inputs, which leads to implementation-defined and possibly undefined behaviour.

$$\mathsf{uint64\_MINMAX}(a,b) = \min(a,b),\ \ \max(a,b) \iff$$
$$\left(\max(a,b) - \min(a,b) < 2^{63}\right) \vee \left((b < a) \wedge \left(a - b = 2^{63}\right)\right)$$

Fig. 2: Conditions for uint64_MINMAX to behave correctly.

The problem in both cases is the most significant bit; the sign bit in the signed case. Using a Cryptol implementation of the unsigned case, and careful, iterative use of its SMT-solving capabilities, we were able to isolate a condition (Fig. 2) for the first macro to work correctly. That is, the first macro only works when the inputs have the same most significant bit.

We did not detect the bug with the second macro Cryptol directly, but instead when verifying the sorting algorithm in SAW. This is because Cryptol's behaviour for integer overflow is always that it wraps, which is also the assumption of the macro. However, SAW's symbolic execution models the C specification itself, in which it is implementation-defined whether signed integers are treated as modular or if they overflow, and an overflow is undefined behaviour. Therefore, SAW detects the possibility of undefined behaviour and terminates the symbolic execution. The condition for avoiding any possibility of overflow is exactly that $b - a$ can be stored in a 32-bit integer, which admits a similar condition to that for the 64-bit case.

By looking at where these functions are used and what is stored in them, it appears that the troublesome bit will only ever be set to 0 in practice, and thus these bugs appear to be of limited, if at all, impact to Classic McEliece. However, another implementor might re-use these sorting functions in another

context. Indeed, these functions are derived from a separate library, the `djbsort` library [11], and thus the bugs could have a wider impact. These functions could be fixed by applying input validation to check for the problematic cases.

This sort of subtle issue is one that is very hard to see with casual human review, but was easy to spot once the formal tool drew our attention to it. Thus this example further motivates the use of formal verification tools in the development of cryptographic schemes.

4 Verifying aspects of Classic McEliece with Lean

In this section, we report on our efforts to apply the interactive theorem prover Lean 3 and its mathlib library [27] to produce verified proofs relating to Classic McEliece. Lean aims to be both a functional programming language, in which it is easy to write correct and maintainable code; and also an interactive theorem prover, similar to Coq [17]. Like Coq and its MathComp library, Lean has its own mathematical components library, mathlib. The mathlib library has enjoyed extensive interest from the pure mathematics community and has been growing and updating at a rapid pace in the last few years. The style in Lean and mathlib is generalised and abstract; in contrast to Coq formalisations, which tend towards a concrete approach [7].

Verification of control bit constructions In [13], Bernstein uses the HOL Light tool to gives proofs, and formal verifications of those proofs, for the control bit constructions used in Classic McEliece. As a proof of concept, we decided to attempt to re-implement the same proofs in Lean. The proofs we were able to obtain verified more theorems than the verifications in [13]; in addition, unlike HOL Light, it is relatively easy in Lean to talk about permutations of $\{0, 1, ..., n\}$ rather than permutations of $\{0, 1, ...\}$ that fix $\{n, n+1, ..., \}$, or indeed permutations defined on any well-ordered type. As such, unlike the verifications in HOL Light, the theorems we verified were closer to the original mathematical statements, with no translations required.

However, these proofs are not compatabile in the most recent version of Lean and mathlib, because the library itself has advanced since then. While they are available [2], they would require further work to update to the latest version of the library. This illustrates an issue with formal methods in general and ones based on unstable libraries in particular: they create an extra technical debt as they require maintenance. Nevertheless, the relative ease at which our work proceeded made us optimistic to try more experiments in proving aspects of Classic McEliece's design using Lean in future.

Verification of coding theory We also investigated the use of Lean to verify the correctness of the decoding methods used in Classic McEliece. We targeted a recent monograph of Bernstein [12] setting out the necessary theorems of coding theory used in the proof of correctness. In theory, Lean's mathlib contains all

the mathematics necessary to prove these theorems. In practice, there were a number of hurdles to overcome, for example in edge cases that need to be formally specified even though they may appear not to matter on paper, and this work is still in progress.

Towards this goal, we have provided to mathlib a refactor of theorems about Lagrange interpolation, and an implementation of the Hamming distance and theorems around it. These theorems are a building block towards verifying Goppa codes and Classic McEliece. Given the recent publishing of a Lean 4 specification of Classic McEliece [3], the possibility (even if a difficult one) of joining this work with proofs about the abstract design of the scheme is an interesting and potentially exciting one. This would be 'first strand' verification — in that it is verifying the mathematical correctness of the design itself — combined with 'second strand' verification — as it would be verifying an implementation against the constraints of a design.

5 Conclusions and perspectives

5.1 Recommendations

Our work adds to the increasing body of work showing that formal approaches can and should be incorporated into cryptographic design evaluation. In particular, we demonstrate that it is meaningful to talk of the verifiability of a particular implementation, and, by extension, of a particular design. Such verifiability can be treated as evidence in favour of a proposed design or implementation. Advocates for the use of computer-aided cryptography should aim to play a role in the setting of common standards around verifiability.

Designers and implementers of cryptographic schemes can follow two main approaches to incorporate computer-aided cryptography techniques. The first approach is using tools that aim to verify existing code. In this case, we recommend engaging with the limits of the chosen tool before beginning implementation. For instance, with SAW/Cryptol, we found that large loops that carried large amounts of state between each loop iteration were not feasibly verifiable. Other tools will have different limitations. Such tools can be incorporated into the software development cycle, as prior work has demonstrated [19, 23]. It is easier to adjust the design of an implementation at an earlier stage in its development, and our experience in this study demonstrates that seemingly-small alterations in a design can make a real difference to verifiability.

The second approach is for implementors to choose a synthesis-first approach [9, 18]. Novel cryptography has the advantage that it is necessarily based on "fresh" code, and has the flexibility to support verified code synthesis. Moreover, such implementations can be as efficient as hand-written code, if not more so.

5.2 Future work using similar approaches

Verification of Classic McEliece The functions that we could verify or partially verify with SAW/Cryptol constitute the core of the decryption and en-

cryption operations of Classic McEliece. It would be interesting to extend this to the higher-level encapsulation or decapsulation functions. For example, their memory safety could be determined by checking that SAW could symbolically execute them. In addition, some components of the implementation were technically challenging to specify in Cryptol. For instance, whilst we were able to produce a specification for the function that calculates permutation control bits, we were not successful at using it in verification. Prior work has explored formally proving the design of the formulae used for the control bit calculation [13]. Future work should seek to explore this further, perhaps porting these proofs to a language like Project Everest's F* [18] which has the facility for code synthesis.

Verification using Lean Lean is a relatively new theorem prover, and as such has seen relatively little attention from the cryptographic community. Lean's strong support for Unicode and the tendency of mathlib towards abstraction and generality means that statements and even proofs in mathlib can look closer to their "pen and paper" equivalents in, say, Coq. This is of interest for forms of cryptography in which the underlying constructions are often abstract and mathematical, even though the instantiations are necessarily concrete. The rapid development of Lean and mathlib means there is an opportunity for cryptographers to "get in at the ground floor" and shape the implementation choices behind key concepts. We see this as an important direction for future work.

References

1. Announcing four candidates to be standardized, plus fourth round candidates: CSRC. `https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4`. NIST [Accessed September 07 2022].
2. Control Bits Verification. `https://github.com/linesthatinterlace/verif-cb`. Wrenna Robson [Accessed September 12 2022].
3. Cryptography in Lean 4. `https://github.com/joehendrix/lean-crypto`. Joe Hendrix [Accessed September 7 2022].
4. Cryptol-Specs. `https://github.com/GaloisInc/cryptol-specs`. Galois Incorporated [Accessed January 7 2022].
5. Post-quantum cryptography: CSRC. `https://csrc.nist.gov/projects/post-quantum-cryptography`. NIST [Accessed January 18 2022].
6. PQC Verification. `https://github.com/linesthatinterlace/pqc-verification`. Wrenna Robson [Accessed September 7 2022].
7. Reynald Affeldt. A Coq formalization of information theory and linear error correcting codes. `https://github.com/affeldt-aist/infotheo`, August 2022.
8. Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, et al. Classic McEliece: conservative code-based cryptography. `https://classic.mceliece.org/nist/mceliece-20201010.pdf`, October 2020.
9. José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. The last mile: High-assurance and high-speed cryptographic implementations. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 965–982. IEEE, 2020.

10. Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-aided cryptography. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 777–795. IEEE, 2021.
11. Daniel J. Bernstein. djbsort. `https://sorting.cr.yp.to`, November 2019.
12. Daniel J. Bernstein. Understanding binary-Goppa decoding. `https://cr.yp.to/papers/goppadecoding-20220320.pdf`, November 2019.
13. Daniel J. Bernstein. Verified fast formulas for control bits for permutation networks. `https://ia.cr/2020/1493`, 2020. Cryptology ePrint Archive, Report 2020/1493.
14. Daniel J. Bernstein. Fast verified post-quantum software. International Cryptographic Module Conference 2021, 2021.
15. Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, Chitchanok Chuengsatiansup, Tanja Lange, Adrian Marotzke, Bo-Yuan Peng, Nicola Tuveri, Christine van Vredendaal, and Bo-Yin Yang. NTRU Prime: round 3. `https://ntruprime.cr.yp.to/nist/ntruprime-20201007.pdf`, October 2020.
16. Daniel J Bernstein, Tung Chou, and Peter Schwabe. McBits: fast constant-time code-based cryptography. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 250–272. Springer, 2013.
17. Y. Bertot, G. Huet, P. Castéran, and C. Paulin-Mohring. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg, 2013.
18. Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch, et al. Everest: Towards a verified, drop-in replacement of HTTPS. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
19. Brett Boston, Samuel Breese, Joey Dodds, Mike Dodds, Brian Huffman, Adam Petcher, and Andrei Stefanescu. Verified cryptographic code for everybody. In *International Conference on Computer Aided Verification*, pages 645–668. Springer, 2021.
20. Kyle Carter, Adam Foltzer, Joe Hendrix, Brian Huffman, and Aaron Tomb. SAW: the software analysis workbench. In *Proceedings of the 2013 ACM SIGAda annual conference on High integrity language technology*, pages 15–18, 2013.
21. Wouter Castryck and Thomas Decru. An efficient key recovery attack on sidh (preliminary version). Cryptology ePrint Archive, Paper 2022/975, 2022. `https://eprint.iacr.org/2022/975`.
22. Tung Chou. McBits revisited. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 213–231. Springer, 2017.
23. Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Serdar Tasiran, et al. Continuous formal verification of Amazon s2n. In *International Conference on Computer Aided Verification*, pages 430–446. Springer, 2018.
24. Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic — with proofs, without compromises. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1202–1219. IEEE, 2019.
25. Levent Erkök, Magnus Carlsson, and Adam Wick. Hardware/software coverification of cryptographic algorithms using Cryptol. In *2009 Formal Methods in Computer-Aided Design*, pages 188–191. IEEE, 2009.

26. Yu-Fu Fu, Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Signed cryptographic program verification with typed cryptoline. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1591–1606, 2019.
27. The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 367–381, New York, NY, USA, 2020. Association for Computing Machinery.
28. Robert J McEliece. A public-key cryptosystem based on algebraic coding theory. *DSN Progress Report*, 4244:114–116, 1978.
29. Nicky Mouha and Asmaa Hailane. The application of formal methods to real-world cryptographic algorithms, protocols, and systems. *Computer*, 54(01):29–38, 2021.
30. Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*, pages 625–635. Springer, 2021.
31. Wrenna Robson. Classic McEliece Verification. `https://github.com/linesthatinterlace/pqc-verification`, January 2022.
32. Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1806, 2017.